CS 267 Applications of Parallel Computers

Lecture 5: More about Distributed Memory Computers and Programming

David H. Bailey

Based on previous notes by James Demmel and David Culler

http://www.nersc.gov/~dhbailey/cs267

Recap of Last Lecture

- **Shared memory processors**
  - **Caches in individual processors must be kept coherent -- multiple cached copies of same location must be kept equal.**
  - **Requires clever hardware (see CS258).**
  - **Distant memory much more expensive to access.**
- **Shared memory programming**
  - **Starting, stopping threads.**
  - **Synchronization with barriers, locks.**
  - **OpenMP is the emerging standard for the shared memory parallel programming model.**

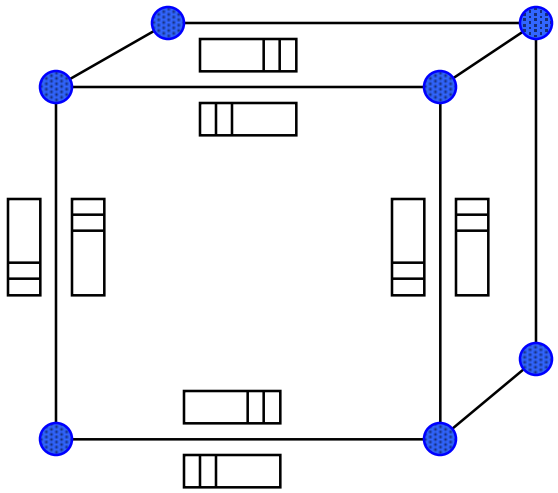
Outline

- **Distributed Memory Architectures**
  - Topologies
  - Cost models
- **Distributed Memory Programming**
  - Send and receive operations
  - Collective communication
- **Sharks and Fish Example**
  - Gravity

History and Terminology

Historical Perspective

- **Early machines were:**
  - **Collection of microprocessors.**
  - **Communication was performed using bi-directional queues between nearest neighbors.**
- **Messages were forwarded by processors on path.**
- **There was a strong emphasis on topology in algorithms, in order to minimize the number of hops.**



Network Analogy

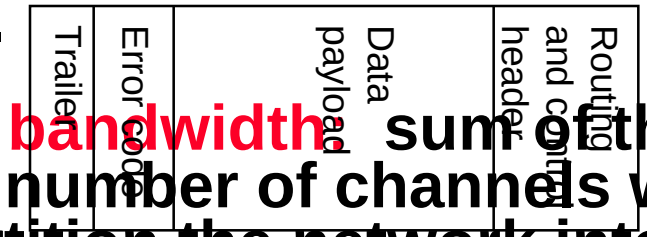
- To have a large number of transfers occurring at once, you need a large number of distinct wires.
- Networks are like streets:
  - Link = street.
  - Switch = intersection.
  - Distances (hops) = number of blocks traveled.
  - Routing algorithm = travel plan.
- Properties:
  - Latency: how long to get between nodes in the network.
  - Bandwidth: how much data can be moved per unit time:
  - Bandwidth is limited by the number of wires and the rate at which each wire can accept data.

Characteristics of a Network

- **Topology (how things are connected)**
  - **Crossbar, ring, 2-D and 2-D torus, hypercube, omega network.**
- **Routing algorithm:**
  - **Example: all east-west then all north-south (avoids deadlock).**
- **Switching strategy:**
  - **Circuit switching: full path reserved for entire message, like the telephone.**
  - **Packet switching: message broken into separately-routed packets, like the post office.**
- **Flow control (what if there is congestion):**
  - **Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.**

Properties of a Network

- **Diameter**: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- A network is **partitioned** into two or more disjoint sub-graphs if some nodes cannot reach others.
- The **bandwidth** of a link = w * 1/t
  - w is the number of wires
  - t is the time per bit
- **Effective bandwidth** is usually lower due to packet overhead.



- **Bisection bandwidth**: sum of the bandwidths of the minimum number of channels which, if removed, would partition the network into two sub-graphs.

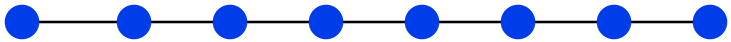
Network Topology

- In the early years of parallel computing, there was considerable research in network topology and in mapping algorithms to topology.
- Key cost to be minimized in early years: number of "hops" (communication steps) between nodes.
- Modern networks hide hop cost (ie, "wormhole routing"), so the underlying topology is no longer a major factor in algorithm performance.
- Example: On IBM SP system, hardware latency varies from 0.5 usec to 1.5 usec, but user-level message passing latency is roughly 36 usec.

However, since some algorithms have a natural topology, it is worthwhile to have some background in this arena.

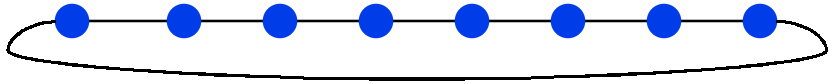
Linear and Ring Topologies

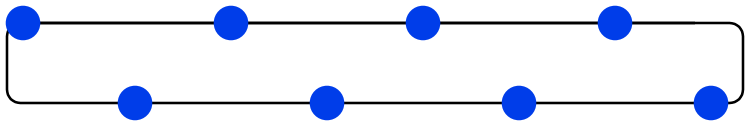
° Linear array



- Diameter = n-1; average distance ~ n/3.
- Bisection bandwidth = 1.

° Torus or Ring



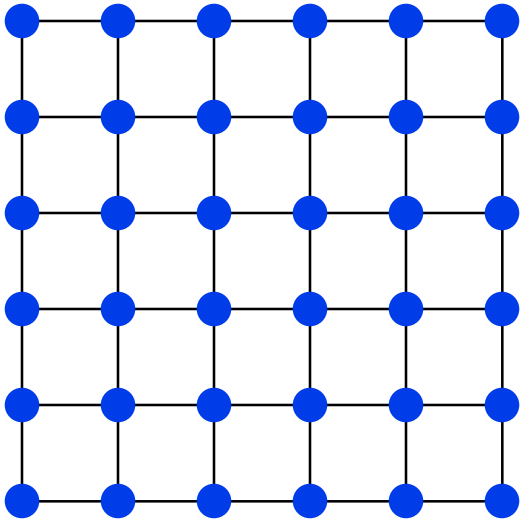


- Diameter = n/2; average distance ~ n/4.
- Bisection bandwidth = 2.
- Natural for algorithms that work with 1D arrays.

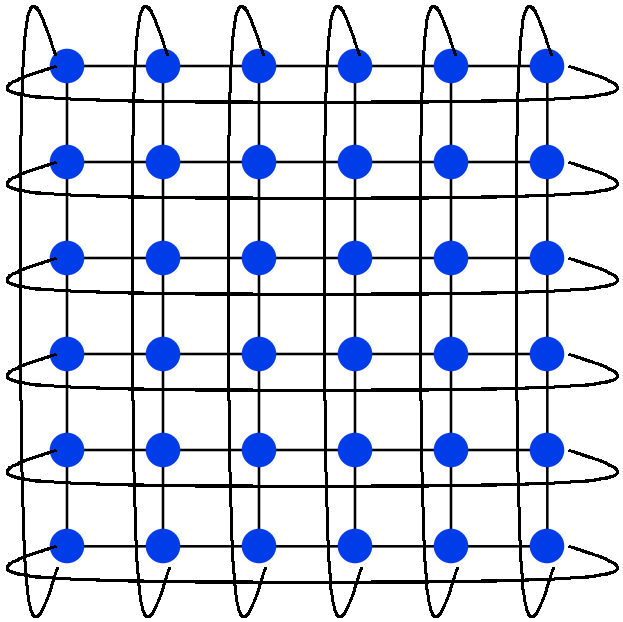
Meshes and Tori

- 2D
  - Diameter = $2\sqrt{n}$
  - Bisection bandwidth = $\sqrt{n}$

2D mesh



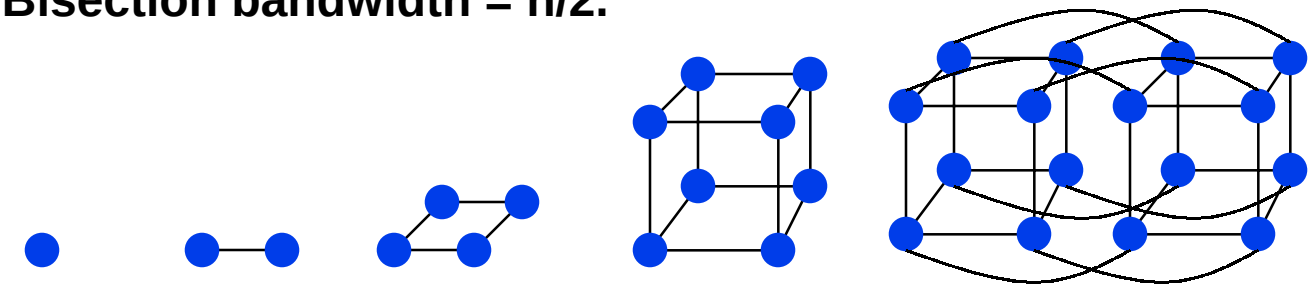
2D torus



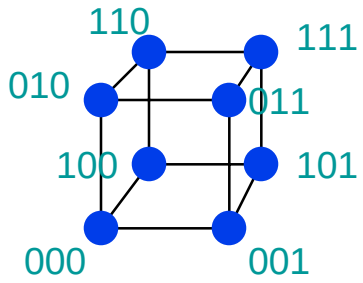
- Often used as network in machines.
- Generalizes to higher dimensions (Cray T3D used 3D Torus).
- Natural for algorithms that work with 2D and/or 3D arrays.

Hypercubes

- Number of nodes n = 2^d for dimension d.
  - Diameter = d.
  - Bisection bandwidth = n/2.

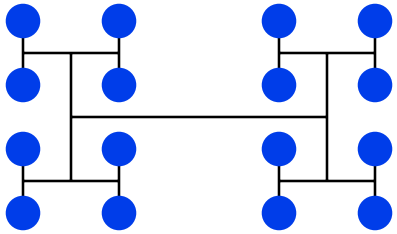


- Popular in early machines (Intel iPSC, NCUBE).
  - Lots of clever algorithms.
  - See 1996 notes.
- Greycode addressing:
  - Each node connected to d others with 1 bit different.

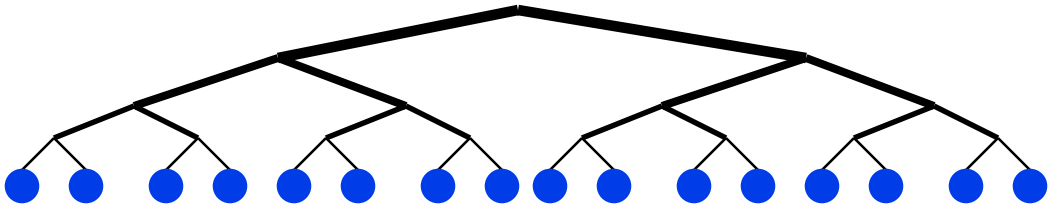


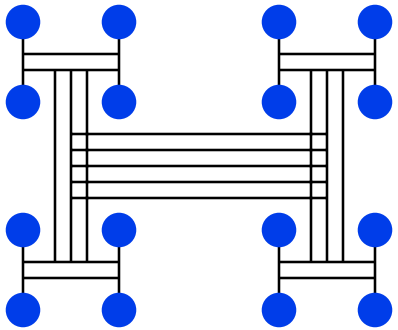
Trees

- Diameter = log n.
- Bisection bandwidth = 1.
- Easy layout as planar graph.



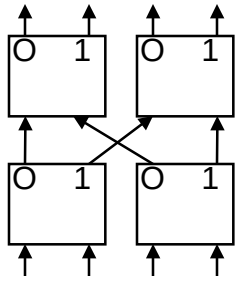
- Many tree algorithms (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.
  - Example: Thinking Machines CM-5.

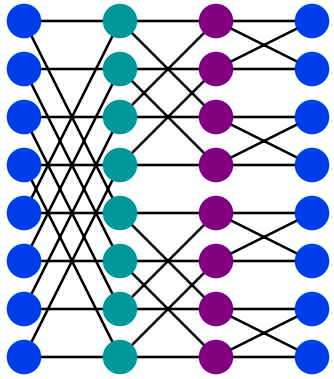




Butterflies

- Diameter = log n.
- Bisection bandwidth = n.
- Cost: lots of wires.
- Used in BBN Butterfly.
- Natural for FFT.





Evolution of Distributed Memory Multiprocessors

- **Special queue connections are being replaced by direct memory access (DMA):**
  - **Processor packs or copies messages.**
  - **Initiates transfer, goes on computing.**
- **Message passing libraries provide store-and-forward abstraction:**
  - **Can send/receive between any pair of nodes, not just along one wire.**
  - **Time proportional to distance since each processor along path must participate.**
- **Wormhole routing in hardware:**
  - **Special message processors do not interrupt main processors along path.**
  - **Message sends are pipelined.**
  - **Processors don’t wait for complete message before forwarding.**

Performance Models

PRAM

- Parallel Random Access Memory.
- All memory access operations complete in one clock period -- no concept of memory hierarchy (“too good to be true”).
- OK for understanding whether an algorithm has enough parallelism at all.
- Slightly more realistic: Concurrent Read Exclusive Write (CREW) PRAM.

Latency and Bandwidth Model

- Time to send message of length n is roughly.

$$\text{Time} = \text{latency} + n*\text{cost\_per\_word}$$
$$= \text{latency} + n/\text{bandwidth}$$

- Topology is assumed irrelevant.
- Often called "α–β model" and written

$$\text{Time} = \alpha + n*\beta$$

- Usually $\alpha >> \beta >>$ time per flop.
  - One long message is cheaper than many short ones.

$$\alpha + n*\beta << n*(\alpha + 1*\beta)$$

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: Need large computation-to-communication ratio to be efficient.

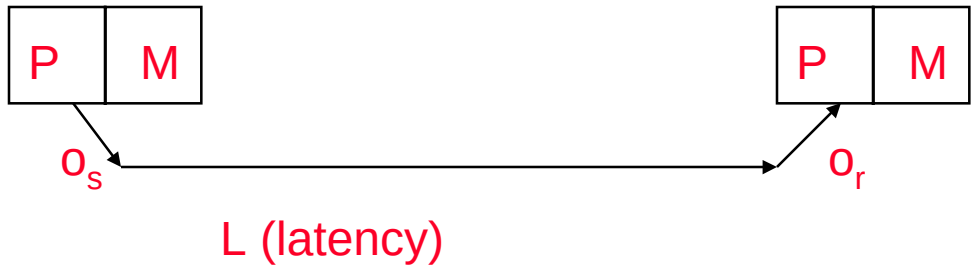
Example communication costs

- **α and β measured in units of flops, β measured per 8-byte word**

Machine	Year	α	β	Mflop rate per proc
CM-5	1992	1900	20	20
IBM SP-1	1993	5000	32	100
Intel Paragon	1994	1500	2.3	50
IBM SP-2	1994	7000	40	200
Cray T3D (PVM)	1994	1974	28	94
UCB NOW	1996	2880	38	180
SGI Power Challenge	1995	3080	39	308
SUN E6000	1996	1980	9	180

A more detailed performance model: LogP

- L: latency across the network.
- o: overhead (sending and receiving busy time).
- g: gap between messages (1/bandwidth).
- P: number of processors.



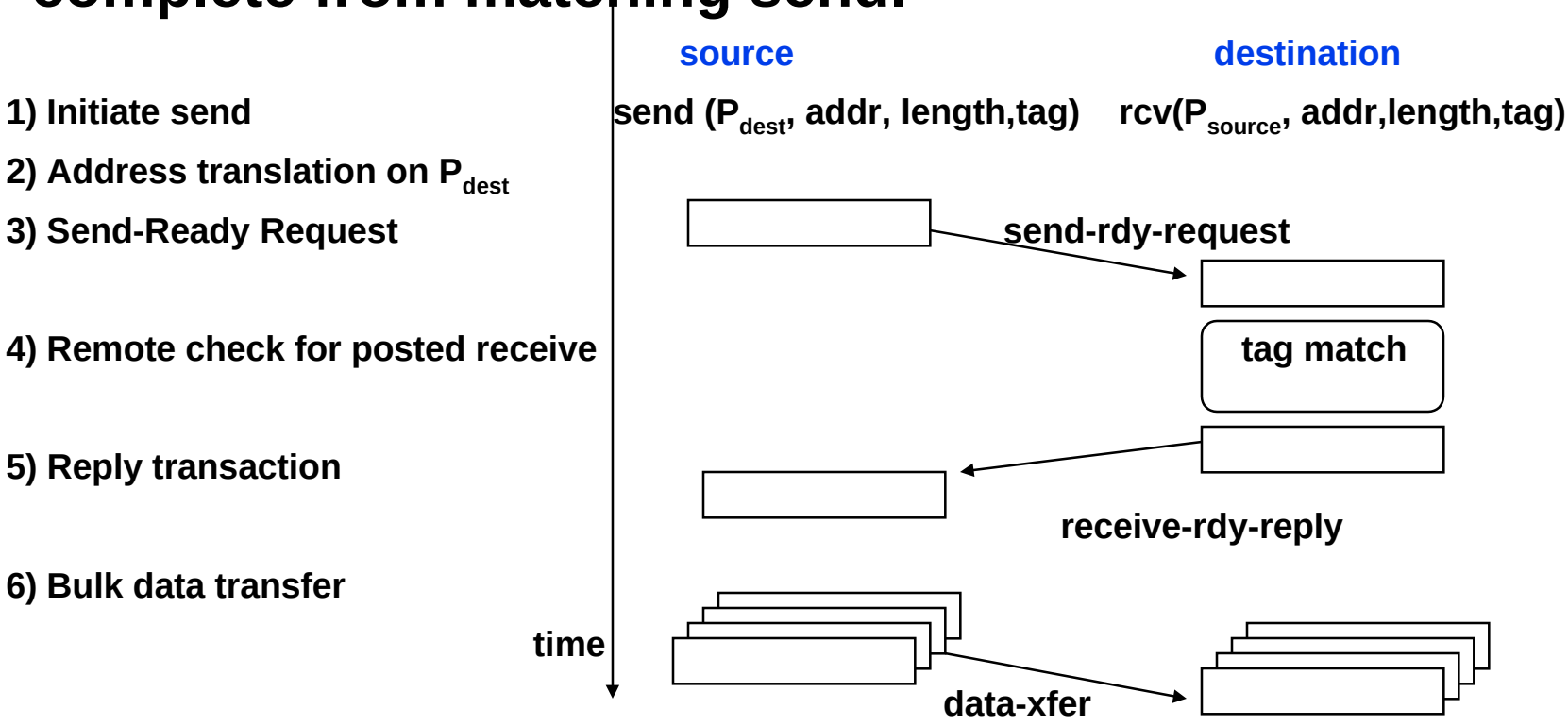
- People often group overheads into latency (α, β model).
- Real costs more complicated -- see Culler/Singh, Chapter 7.

Message Passing Libraries

- Many “message passing libraries” available
  - Chameleon, from ANL.
  - CMMD, from Thinking Machines.
  - Express, commercial.
  - MPL, native library on IBM SP-2.
  - NX, native library on Intel Paragon.
  - Zipcode, from LLL.
  - PVM, Parallel Virtual Machine, public, from ORNL/UTK.
  - Others...
  - MPI, Message Passing Interface, now the industry standard.
- Need standards to write portable code.
- Rest of this discussion independent of which library.
- Will have a detailed MPI lecture later.

Implementing Synchronous Message Passing

- Send operations complete after matching receive and source data has been sent.
- Receive operations complete after data transfer is complete from matching send.



Example: Permuting Data

° **Exchanging data between Procs 0 and 1, V.1: What goes wrong?**

Processor 0
send(1, item0, 1, tag1)
recv(1, item1, 1, tag2)

Processor 1
send(0, item1, 1, tag2)
recv(0, item0, 1, tag1)

° **Deadlock**

° **Exchanging data between Proc 0 and 1, V.2:**

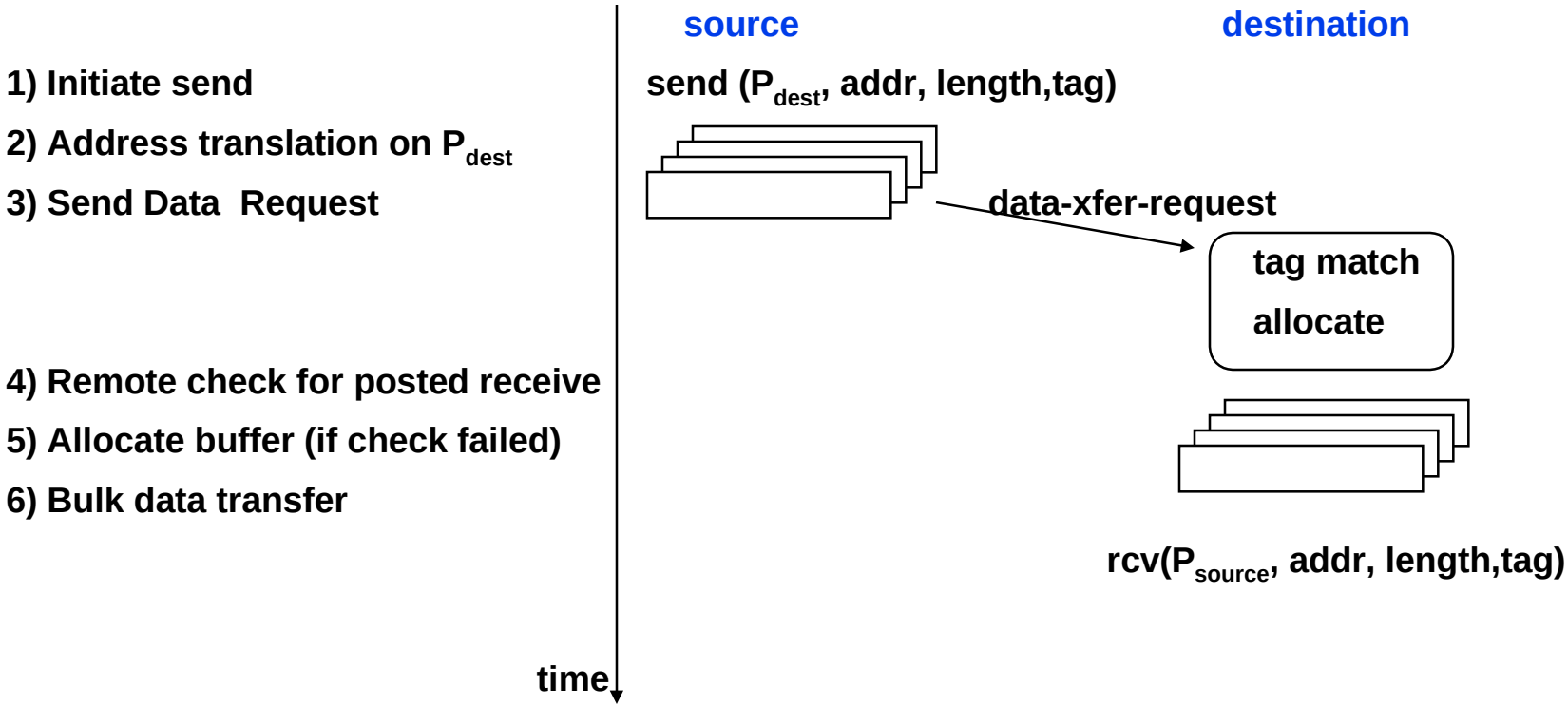
Processor 0
send(1, item0, 1, tag1)
recv(1, item1, 1, tag2)

Processor 1
recv(0, item0, 1, tag1)
send(0,item1, 1, tag2)

° **What about a general permutation, where Proc j wants to send to Proc s(j), where s(1),s(2),…,s(P) is a permutation of 1,2,…,P?**

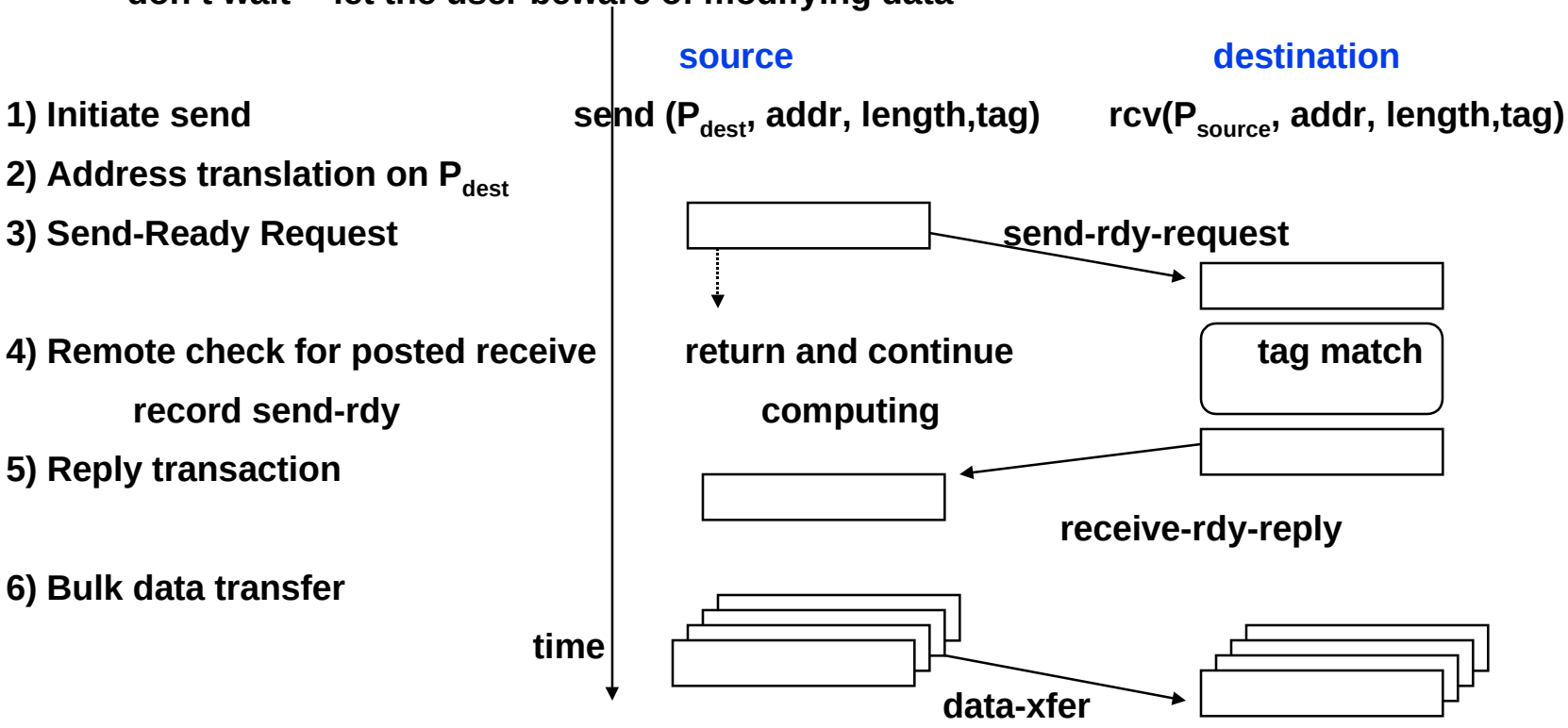
Implementing Asynchronous Message Passing

° **Optimistic single-phase protocol assumes the destination can buffer data on demand.**



Safe Asynchronous Message Passing

- Use 3-phase protocol
- Buffer on sending side
- Variations on send completion
  - wait until data copied from user to system buffer
  - don't wait -- let the user beware of modifying data



Example Revisited: Permuting Data

° **Processor j sends item to Processor s(j), where s(1),…,s(P) is a permutation of 1,…,P**

```
Processor j
    send_asynch(s(j), item, 1, tag)
    recv_block( ANY,  item, 1, tag)
```

° **What could go wrong?**
° **Need to understand semantics of send and receive.**
° **Many flavors available.**

Other operations besides send/receive

- "Collective Communication" (more than 2 procs)
  - Broadcast data from one processor to all others.
  - Barrier.
  - Reductions (sum, product, max, min, boolean and, #, …), where # is any "associative" operation.
  - Scatter/Gather.
  - Parallel prefix -- Proc j owns x(j) and computes y(j) = x(1) # x(2) # … # x(j).
  - Can apply to all other processors, or a user-define subset.
  - Cost = O(log P) using a tree.
- Status operations
  - Enquire about/Wait for asynchronous send/receives to complete.
  - How many processors are there?
  - What is my processor number?

Example: Sharks and Fish

- N fish on P procs, N/P fish per processor
  - At each time step, compute forces on fish and move them
- Need to compute gravitational interaction
  - In usual n^2 algorithm, every fish depends on every other fish.
  - Every fish needs to “visit” every processor, even if it “lives” on just one.
- What is the cost?

Two Algorithms for Gravity: What are their costs?

Algorithm 1

```
Copy local Fish array of length N/P to Tmp array
for j = 1 to N
    for k = 1 to N/P,   Compute force of Tmp(k) on Fish(k)
    "Rotate" Tmp by 1
        for k=2 to N/P, Tmp(k) <= Tmp(k-1)
        recv(my_proc - 1,Tmp(1))
        send(my_proc+1,Tmp(N/P)
```

Algorithm 2

```
Copy local Fish array of length N/P to Tmp array
for j = 1 to P
    for k=1 to N/P, for m=1 to N/P, Compute force of Tmp(k) on Fish(m)
    "Rotate" Tmp by N/P
        recv(my_proc - 1,Tmp(1:N/P))
        send(my_proc+1,Tmp(1:N/P))
```

What could go wrong? (be careful of overwriting Tmp)

More Algorithms for Gravity

- **Algorithm 3 (in sharks and fish code):**
  - **All processors send their Fish to Proc 0.**
  - **Proc 0 broadcasts all Fish to all processors.**
- **Tree-algorithms:**
  - **Barnes-Hut, Greengard-Rokhlin, Anderson.**
  - **O(N log N) instead of O(N^2).**
  - **Parallelizable with cleverness.**
  - **“Just” an approximation, but as accurate as you like (often only a few digits are needed, so why pay for more).**
  - **Same idea works for other problems where effects of distant objects becomes “smooth” or “compressible”:**
    - **electrostatics, vorticity, …**
    - **radiosity in graphics.**
    - **anything satisfying Poisson equation or something like it.**